

Carnegie-Mellon University
Software Engineering Institute

Distributed Ada Real-Time Kernel

Judy Bamberger
Currie Colket
Robert Firth
Daniel Klein
Roger Van Scoy

August 1988

ADA199482

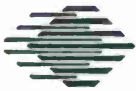
Technical Report

CMU/SEI-88-TR-17

ESD-TR-88-18

August 1988

Distributed Ada Real-Time Kernel



Judy Bamberger

Currie Colket

Robert Firth

Daniel Klein

Roger Van Scoy

Distributed Ada Real-Time Kernel

Approved for public release.
Distribution unlimited.

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This technical report was prepared for the

SEI Joint Program Office
ESD/AVS
Hanscom AFB, MA 01731

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER


Dan Burton
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

Copyright © 1988 Carnegie Mellon University

This document is available through the Defense Technical Information Center. DTIC provides access to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential contractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center, Attn: FDRA, Cameron Station, Alexandria, VA 22304-6145.

Copies of this document are also available through the National Technical Information Services. For information on ordering, please contact NTIS directly: National Technical Information Services, U.S. Department of Commerce, Springfield, VA 22161.

Use of any other trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

Table of Contents

1. Rationale	1
2. System Models	3
2.1. ISO Model	3
2.1.1. Physical Layer	3
2.1.2. Data Link Layer	3
2.1.3. Network Layer	4
2.1.4. Transport Layer	4
2.1.5. Session Layer	4
2.1.6. Presentation Layer	5
2.1.7. Application Layer	5
2.2. Network Model	5
2.3. Process Model	8
2.4. Communication Model	10
2.5. Scheduling Model	10
3. Status of Work	13
4. Conclusions	15
5. Acknowledgement	17
References	19
Appendix A. Summary of Kernel Primitives	21
A.1. Network Management	21
A.2. Process Management	21
A.3. Semaphore Management	21
A.4. Schedule Management	21
A.5. Communication Management	22
A.6. Interrupt Management	22
A.7. Time Management	22
A.8. Alarm Management	23
A.9. Tool Interface	23

List of Figures

Figure 2-1: Network View	6
Figure 2-2: Load Image Creation	7
Figure 2-3: Main Unit Outline	9
Figure 2-4: Process State Diagram	12



Distributed Ada Real-Time Kernel

Abstract. This paper addresses two distinct needs of real-time applications: distribution and hard real-time scheduling mechanisms. Specifically, this paper rejects both the notion of modifying the Ada language to achieve needed real-time solutions and the current fad of extensively modifying the Ada compiler and/or vendor-supplied runtime system. Instead, this paper defines the functionality of a Distributed Ada Real-Time Kernel (hereafter called the Kernel). The goal of the Kernel is to support effectively the execution of distributed, real-time Ada applications in an embedded computer environment by returning control to the user, where it belongs.

1. Rationale

Many people attempting to use Ada in real-time, embedded systems are dissatisfied with the Ada language features defined to provide real-time support and the implementation of those language features by compiler vendors. Currently, this results in:

1. Application-specific tailoring of Ada Runtime Environments (ARTEs) and the addition of compiler-specific **pragmata** to enhance the real-time capabilities of Ada. This leads to projects becoming compiler-dependent. Ada code that depends on compiler and ARTE modifications is no longer portable (often even to an upgraded version of the same vendor's compiler) and is, at best, questionably reusable. Maintenance becomes a nightmare, since it now includes a "one-of-a-kind" product written by a third party (i.e., the vendor-supplied compiler and ARTE). *The consequence of allowing modified ARTEs and pragmata is that the 300 languages in use prior to Ada have been replaced with 3000+ variants of Ada (i.e., one for each Ada project).*
2. Special-interest groups petitioning for additional language features in Ada. This is not the solution either; since all users will not be pleased by any single language change, the "typical" user is targeted by the proposed language change, with the result that NO ONE is satisfied. Once again, the language designers are left in the untenable position of attempting to "second guess" the users' needs. Finally, this approach is of no help for users of Ada in embedded systems *today*. These users need solutions *now*, not in five years when the next version of Ada is due.

What users really want is language functionality, not language features (see [1]). It is imperative to recognize that there are certain areas of functionality that are (and should continue to be) above and beyond the scope of the Ada programming language, or any programming language for that matter. Specifically, it is time to leave the decisions about runtime environments to the people who know best: the applications software and systems engineers who know the intricacies of the system being built. Nor is the Ada language necessarily the appropriate level to express the distribution of an application. Even the Ada Joint Program Office (AJPO) has acknowledged this special case for real-time, embedded systems, as the validation policies now recognize that an "application-specific runtime library" is considered an integral part of the application on a "restricted target" (see [3]), *not* part of the compiler.

Based on these sentiments, our position is that since Ada provides legitimate avenues for the extension of the language (namely the **package** construct), we propose to use this avenue as a

means to handle distribution and real-time scheduling mechanisms. Therefore, the Kernel will provide support for *language functionality* (i.e., the ability to execute Ada programs in a distributed, real-time environment) but it will *not* provide support for *language features* (i.e., Ada tasking primitives).

2. System Models

This chapter describes a number of the models that underlie the entire Kernel.

2.1. ISO Model

The Kernel communication model presents a set of primitives to the user and implements those primitives on an underlying set of distributed processors connected by data paths. The model, the implementation, and the intended mode of use can all be related to the ISO Reference Model (see [5] and [2]).

The ISO Reference Model identifies seven layers, named, from lowest to highest:

1. Physical
2. Data Link
3. Network
4. Transport
5. Session
6. Presentation
7. Application

The target hardware provides the Physical Layer. The Kernel implements the Data Link, Network, and Transport Layers, and therefore presents to the user the Transport Layer. The Kernel thus encapsulates within itself the Data Link and Network Layers, rendering them invisible to the user. Thus, it is at the Transport Layer where errors from lower levels are reported to the application, leaving error recovery in the domain of the application code. The application code can implement the Session, Presentation, and Application Layers, in part by using other Kernel primitives.

2.1.1. Physical Layer

The Physical Layer is represented by the hardware data paths, which support the transmission of a serial bitstream between processors. These hardware data paths are used by the Kernel in a *packet switching* mode; that is, a sequence of bits—a *frame*—is sent at the discretion of the originator, with no implied reservation of resources or preservation of state between frames.

2.1.2. Data Link Layer

This is the layer at which basic error detection and recovery and flow control may be provided. The Kernel uses a simple *datagram* model in which a frame is transmitted with no acknowledgment, no error correction, and no flow control. Minimal error detection is achieved by using a datagram checksum, but any recovery is performed by application code (i.e., above the Transport Layer). Similarly, datagram storage overflow is recognized and reported by the Transport Layer.

2.1.3. Network Layer

In this prototype, the Kernel has a null Network Layer. The Kernel assumes that point-to-point communication is available between any pair of nodes (processors). Routing is accomplished trivially in the sender by dispatching a point-to-point datagram directly to the receiver; no alternative routing is provided.

However, since the abstraction presented to the user is above this layer, a real Network Layer could subsequently be added without requiring any application code to be changed.

2.1.4. Transport Layer

The Kernel builds the Transport Layer by performing physical network connections and subsequent logical-to-physical mappings, actions that together implement the abstraction of direct process-to-process communication by means of messages.

The physical network description is maintained in each processor. The generation of this information is performed by the system/application engineer. Once that information is provided, the Kernel verifies the network connectivity and opens the physical connections between processors.

Subsequently, the logical *processes* and their physical *sites* are communicated to the Kernel. The model on which the Kernel is based assumes that all processes are created at initialization time, that a process never moves, and that a process once dead is never restarted. The Kernel therefore computes the logical-to-physical mapping once only and never subsequently changes it. Attempts to communicate with dead processes are treated as transport errors.

The Transport Layer also performs the conversions between *messages* and the underlying datagrams. In this prototype, this is done trivially by using one datagram per message or per acknowledgement and, if necessary, by restricting the maximum message size accordingly.

The Transport Layer is the layer visible to the user. It supports both unacknowledged send and end-to-end acknowledged send operations. All errors detected in this or any lower layer are reported at this layer, in the form of status codes returned by the Kernel primitives.

2.1.5. Session Layer

This layer is implemented by stylized application code. Since it establishes logical connections between processes (i.e., initializes the data structures required by the Transport Layer), its presence is required, and the user must write specific code to create it. This code is part of the application initialization code; it must be present on every processor and, in Ada terms, must be part of the *Main Unit* on that processor.

The model is one of a set of logical processes, each with a user-defined *name* and each with a single *message port* for the reception of messages from other processes.

The Kernel **declare process** primitive indicates an intent to create or communicate with a given named process. It establishes the mapping between application-level process names and Kernel internal names.

The Kernel **create process** primitive creates the process (i.e., sites the process and creates the execution environment for the process code), establishes its message port, and makes that port available to the network. Thereafter, one process may communicate with another.

2.1.6. Presentation Layer

In the Kernel model, the Presentation Layer performs no transformation of data. Rather, it performs the translation between Ada values—values of user-defined data types—and message values. This is done by stylized application code. The purpose of the Presentation Layer is to establish above the Transport Layer the strong typing of the Ada language by ensuring that communicating processes pass only strongly typed data and do so by referencing a common set of data conversion routines bound to a common Ada data type.

2.1.7. Application Layer

This layer uses the Presentation Layer for whatever purpose the code requires. The model here is of parallel independent threads of control executing Ada code, identifying each other by application-level symbolic names and communicating by passing values of Ada data types.

2.2. Network Model

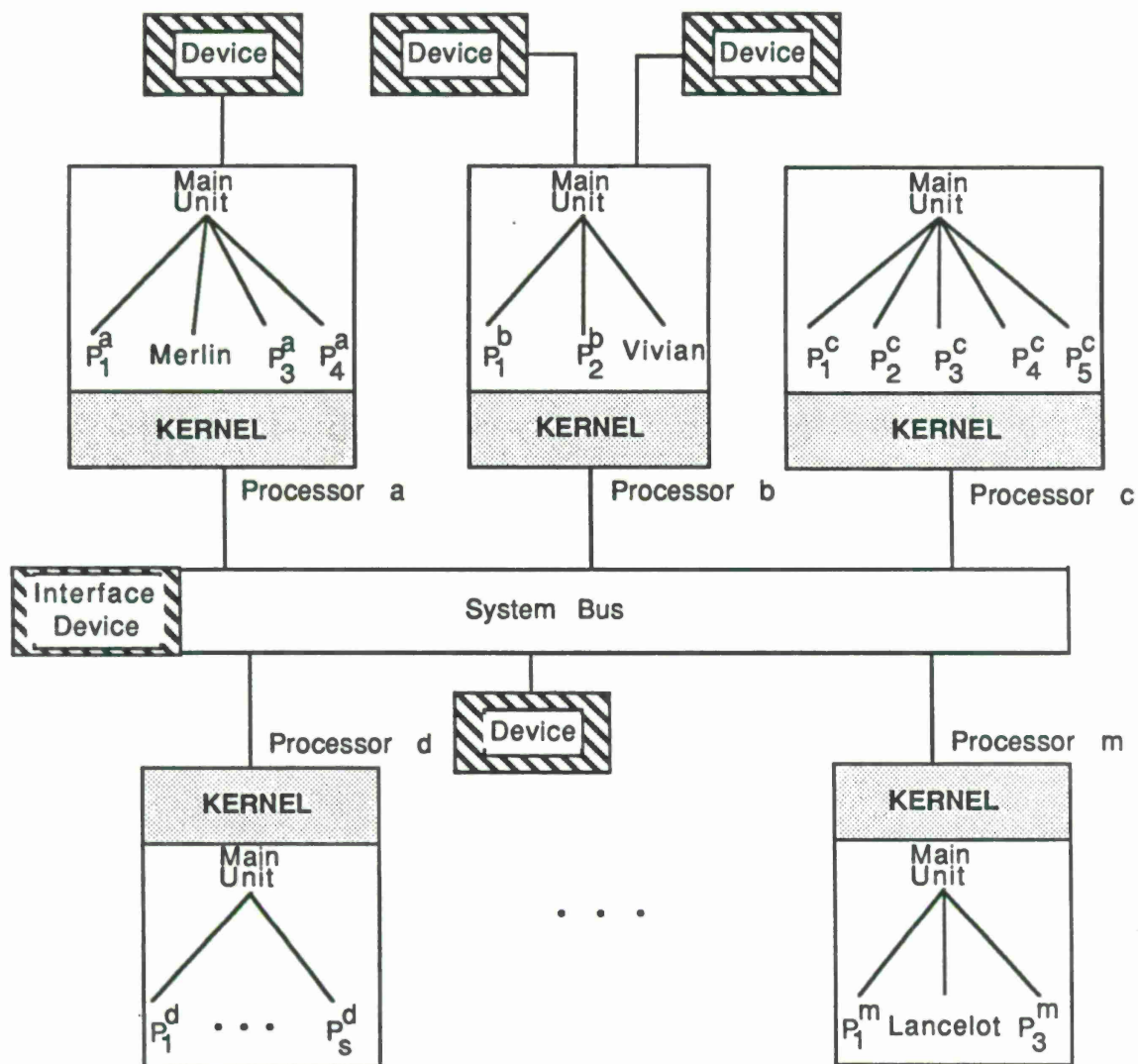
Figure 2-1 presents the view of the network taken by the Kernel.

There are four types of objects the network configuration in which Kernel-based applications operate:

- Processors that are running the Kernel
- Processors (or devices) that are not running the Kernel
- Devices that can interrupt processors
- A system bus connecting all processors

The network configuration is defined by the user and communicated to the Kernel during network initialization. The result of this definition is maintained in a physical mapping table called the Network Configuration Table (NCT).

This model does not assume (or exclude) the existence of shared memory or mass storage devices. Such capabilities would be under the control of the application, not the Kernel, although the Kernel does control the underlying communication medium (i.e., the system bus) and handles *all* inter-process communication over that medium.



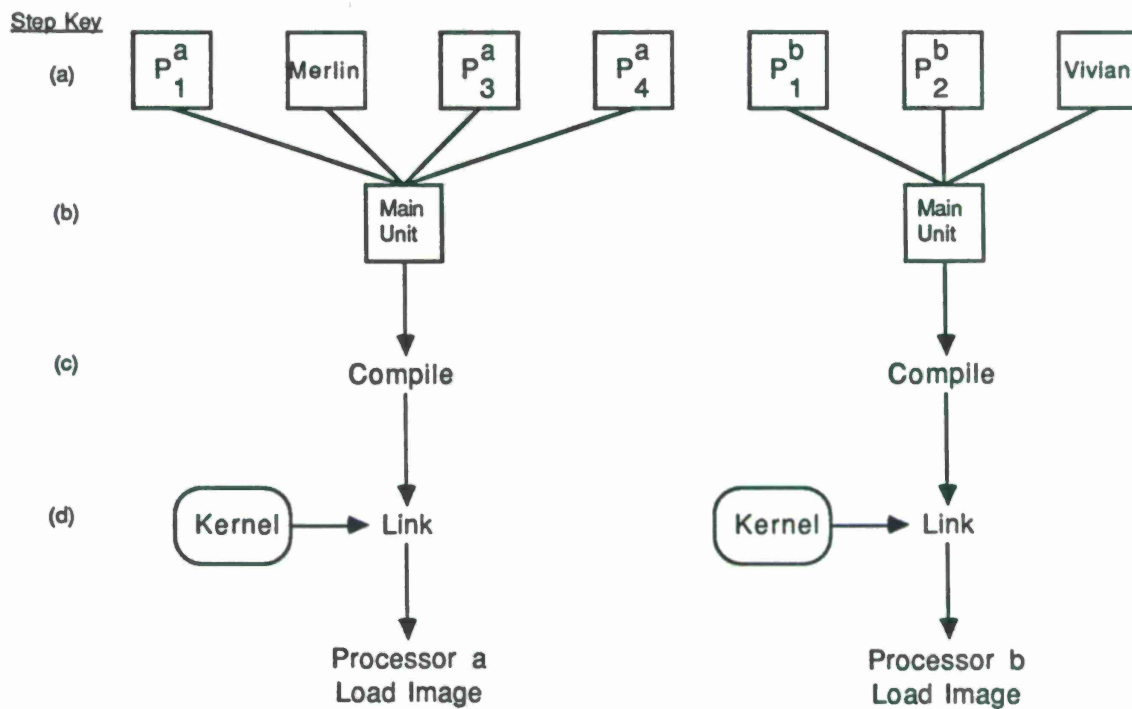
Key

P_i^q : Process # i running on processor q .

Main Unit: The Ada Main Unit running on the processor.

Merlin, Vivian, and Lancelot are named for use in examples.

Figure 2-1: Network View



Key

P_i^q : Process #i running on processor q.

Main Unit: The Ada main unit running on the processor.

Merlin and Vivian are named for use in examples.

Figure 2-2: Load Image Creation

2.3. Process Model

Executing on each processor is the Kernel's general process model. The following are key elements of this model:

1. Each process executes a unit of code, developed as a functional unit.
2. For each processor, the software engineer performs the following steps (illustrated in Figure 2-2):
 - a. Develop the process code.
 - b. Develop the Main Unit for the processor (the function of the Main Unit is explained later).
 - c. Compile the code of the processes and Main Unit.
 - d. Link the Kernel, Main Unit, and processes together to form the load image for that processor.
3. The load image begins execution at the initialization point of the Kernel, which in turn invokes the Main Unit.

When developing a process, the software engineer need not know where the other processes will be located—on a single processor or across multiple processors. The Kernel-supplied communication primitives can be used for all inter-process communication, local or remote, with the Kernel optimizing the local case.

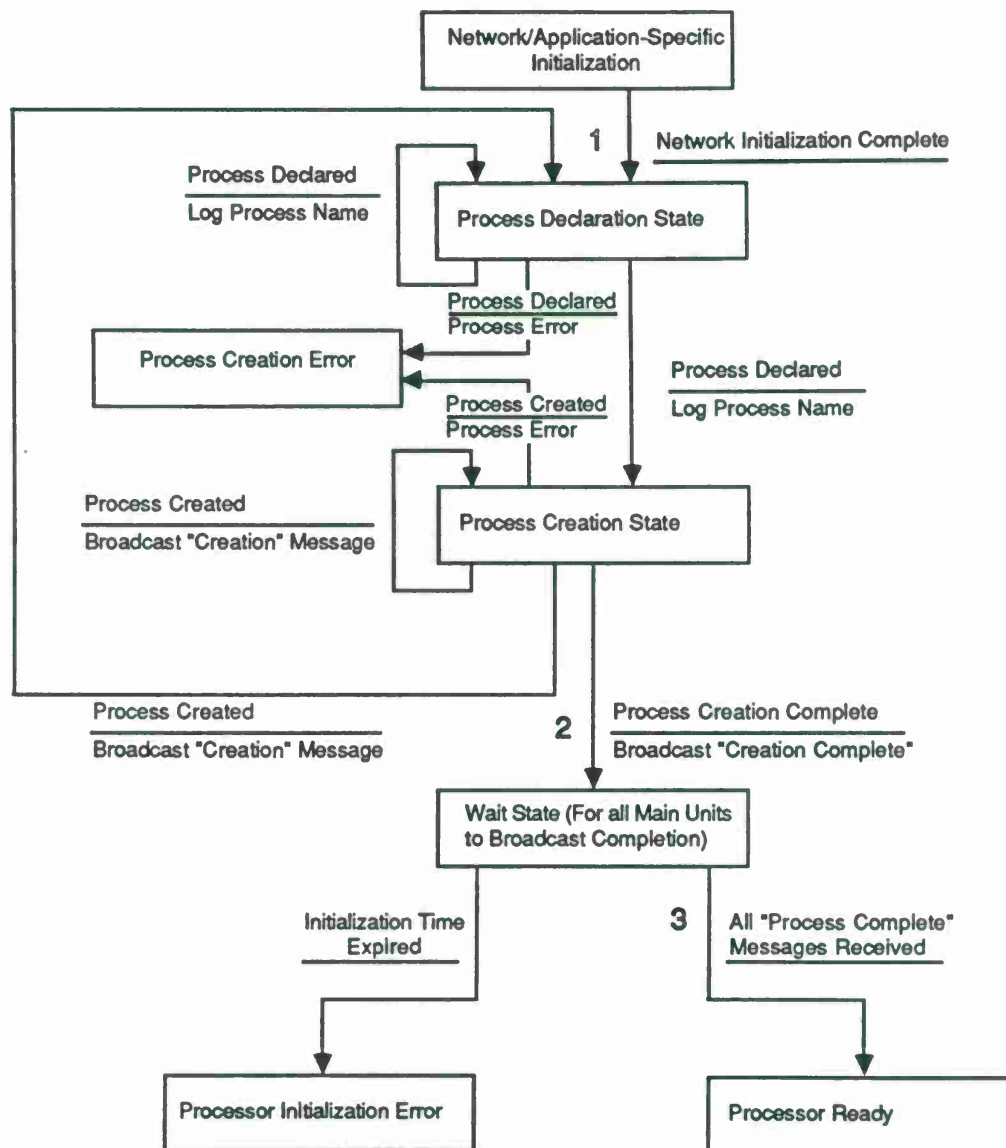
The first application code to execute is the Ada Main Unit. The Main Unit has the responsibility to:

- Define the physical topology of the network (the NCT), as described above.
- *Declare* all processes with which communication is to occur. This information is recorded in the Process Table, which provides the logical topology of the network for the processor.
- *Declare* and *create* the processes to run on this processor, and specify their initial characteristics (preemption, priority, message queue handling, and so forth). This information is also recorded in the Process Table.
- Perform any other system-dependent initialization.

Once this has been accomplished, the Main Unit announces to other processors that all process declaration and creation are complete, and then the Main Unit is descheduled while the processes continue to run independently. The state transition diagram for the Main Unit is shown in Figure 2-3.¹ Figure 2-3 shows the state transition diagram executed by the Main Unit while it is building the logical topology of the processor.

This process initialization protocol is quite simple, perhaps unrealistically so, for many real-time applications. Even after surveying a variety of real-time applications, no consensus of processor and process initialization emerged. This led to treating the initialization issue in two parts:

¹See [4] for a complete description of the state transition diagram notation used in this document.



Key Initialization Points

- 1 All processors know that the physical topology of the network is correct. All clocks are synchronized.
- 2 Processor configuration is complete; waiting for remaining processors to complete.
- 3 All processors know that logical topology of the network is consistent. The application is ready to execute.

Figure 2-3: Main Unit Outline

1. The Kernel would provide a simple mechanism by which to get an application executing.
2. The Kernel would provide a complete, well-defined Interface between the Initialization modules and the rest of the Kernel primitives to allow Kernel users to replace the Kernel's initialization protocol with one more appropriate to their application requirements. This interface is manifest in the NCT, the Process Table constructs, and the primitives shown in the appendix.

2.4. Communication Model

The Kernel provides a minimal set of primitives to support communication among the processes in the network. This was done intentionally to provide a robust, basic set of communication primitives on which any number of more sophisticated communication protocols can be built. All inter-process communication uses a common set of communication primitives—regardless of where the processes are sited (i.e., co-located or remotely located).

The fundamental concepts on which the communication model is built are:

- All communication is point-to-point.
- A sender must specify the receiver.
- All messages are tagged.
- A receiver gets all messages and is told the sender of each.
- A receiver cannot ask to receive only from specific senders.
- Messages do not have priorities.

The following communication primitives are provided:

- *Send*: The capability to do a "blind" send to any other Kernel process or any non-Kernel processor.
- *Send and wait*: The capability to do a synchronized send to another Kernel process, where a Kernel-to-Kernel acknowledgement is returned upon the receiver's acceptance of the message.
- *Receive*: The capability to do a receive of any message from any Kernel process or non-Kernel processor.

The purpose of a message is to convey information between processes. To the Kernel, a message is just a sequence of uninterpreted bits. The Kernel provides the untyped primitives (in the Transport Layer); the users may build on them whatever application-specific functionality is needed (in the Presentation Layer).

2.5. Scheduling Model

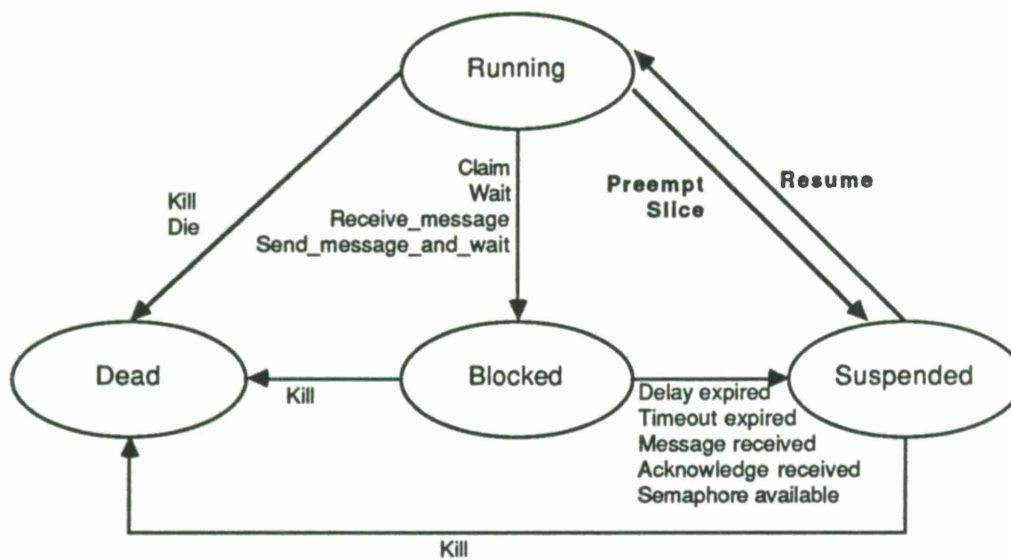
There are four concepts that underlie the scheduling model of the Kernel:

- The *state* of a process. There are only four states in which a process can ever be, as indicated in Figure 2-4. A *running* process is the process currently executing on the processor. A *suspended* process is one that is eligible to be running but, due to

its priority, is not currently running. A *blocked* process is one that is unable to run as it is waiting for an event to occur (see *events* below). A *dead* process is one that is unable to run again; it is terminated.

- *Events* that may cause a change in that state. An event is something that happens to a process, such as the arrival of a message, the arrival of a Kernel-to-Kernel acknowledgement, being killed by another process, and so forth. Events are inherently unpredictable.
- The *time* at which certain events occur. The Kernel allows applications to invoke time-dependent actions. Such an action is a request for a future change of state. After the change of state is made, the process is suspended. Time (elapsed or absolute) is a parameter to many of the Kernel primitives. Time slicing is also supported.
- The *priority* at which a process is maintained. Every process has a priority, which may be changed dynamically by the process. Priorities are relative within a single processor and incommensurable across processors. Priorities are strict and preemptive; higher priorities always shut out lower priorities. (Processes run at the discretion of the Scheduler, whose algorithm is governed only by priority.) Each Kernel primitive that could block provides the capability for the caller to specify the priority at which the process is to be resumed once the process unblocks.

A process is always in a definite *state*. A process may change its state when it invokes a Kernel primitive, as the result of an event, or as a consequence of the passage of time. Some causes of change can occur *unpredictably*, but the Kernel guarantees that the subsequent state change is *deterministic*. A direct consequence of this is that if all events occur at predictable times and all transaction processing is of fixed duration, then the execution of an application using the Kernel is deterministic.



Key

Label: User - initiated action

Label: Kernel-Initiated action

Figure 2-4: Process State Diagram

3. Status of Work

As of this writing, the project developing the Kernel has been underway for eight months. During that time, the following activities have taken place:

1. Project planning (tasks, milestones, budgets, schedules, software development plan) and resource planning (hardware and software).
2. Identification of an application to provide an initial demonstration of the efficacy of the Kernel.
3. Kernel Facilities Definition document produced and extensively reviewed within the Software Engineering Institute (SEI) and by key real-time experts outside the SEI. This document presents a detailed description of the concepts and models underlying the Kernel—concepts and models that have been only introduced in this paper. In addition, the behavioral and performance requirements for each component of the Kernel are identified. The Kernel comprises the following components:
 - a. Processor Management (overviewed in this paper; the creation of the physical topology).
 - b. Process Management (overviewed in this paper; the creation of the logical topology; process declaration and creation).
 - c. Semaphore Management (a Boolean semaphore capability).
 - d. Schedule Management (overviewed in this paper).
 - e. Communication Management (overviewed in this paper).
 - f. Interrupt Management (supporting the capability to use Ada or other code as interrupt handlers).
 - g. Time Management (providing the abstractions of hardware time, elapsed time, and absolute time).
 - h. Alarm Management (allowing processes to set a timer).
 - i. Tool Interface (a read-only capability to monitor Kernel and application activities).
4. System Model Review held. A detailed description of the Kernel was presented to a number of SEI software and real-time systems experts for scrutiny and evaluation. The results of this review have been factored into the Kernel Facilities Definition document.
5. Architectural specification of the Kernel and the Kernel interface to the Ada application program. These activities are currently in progress.

4. Conclusions

This paper proposes one solution to the problem of using Ada in distributed, real-time, embedded applications—one that can readily be accomplished in the near term. This solution is truly in the spirit of Ada—that is, it uses the Ada language features (e.g., packages, subprograms) to provide an adjunct capability for real-time scheduling and inter-process communication. This alternative returns explicit control of scheduling to the application implementor and provides a uniform communication mechanism for supporting distributed processors or a single processor.

Other difficult areas, such as fault tolerance and multi-level security, are not directly addressed in the Kernel definition. We have, however, examined our primitives in light of these and other equally demanding issues, and we believe the Kernel definition to be extensible enough to accommodate future development in these areas.

Our goal is to provide a viable paradigm of near-term support to a wide number of real-time embedded applications currently being required to use Ada for implementation. We believe that the applications builders—not compiler vendors, not language designers—know best the system-level behavior required for their programs. We believe that standardization of such behavior should be provided via a library package interface under the control of the application implementor, *not* via modifications to the Ada language. We believe our strategy and Kernel definition provide this kind of support.

5. Acknowledgement

This report was presented at NAECON '88 in Dayton, Ohio, by the principal authors, Judy Bamberger and Roger Van Scoy.

References

- [1] Firth, Robert.
A Pragmatic Approach to Ada Insertion.
In *Proceedings of the International Workshop on Real-Time Ada Issues*, pages 24-26.
May, 1987.
- [2] A.S. Tanenbaum.
Network Protocols.
Computing Surveys 13:453-489, 1981.
- [3] Ada Joint Program Office.
Validation Procedures and Guidelines.
Technical Report, AJPO, AdalC, 3D139 (1211 S. Fern, C-107), The Pentagon,
Washington, D.C. 20301-3081, 1987.
- [4] Ward, Paul T. and Stephen J. Mellor.
Structured Development for Real-Time Systems.
Yourdon Press, Englewood Cliffs, NJ, 1985.
- [5] H. Zimmermann.
OSI Reference Model - the ISO model of architecture for open systems interconnection.
IEEE Transactions on Communications COM-28:425-432, 1980.

Appendix A: Summary of Kernel Primitives

A.1. Network Management

Initialize Master. Identifies the invoking processor as responsible for network initialization.

Initialize subordinate. Identifies all other processors and instructs each to wait for the "go" command from the Master.

Start subordinates. Issues the "go" command to all subordinate processors (which also signals that the network is properly connected and healthy).

Create network configuration. Defines the physical topology of the network.

A.2. Process Management

Declare a process. The Main Unit on a processor declares all locally executing processes and all remote processes with which communication is desired.

Create a process. The Main Unit on a processor creates all Kernel processes that are to execute on that processor (these may be cyclic or non-cyclic).

Process creation completed. The Main Unit indicates to the Kernel that all process declarations and creations are now complete.

Die. A Kernel process may indicate that it is complete and ready to be descheduled.

Kill. A Kernel process may cause itself or another process to be abnormally terminated.

A.3. Semaphore Management

Claim. The invoking process attempts to claim the semaphore. The claiming process blocks until the semaphore becomes available or the timeout expires.

Release. The invoking process releases a previously claimed semaphore.

A.4. Schedule Management

Set process priority. A process may set or get its own priority.

Set process preemption. A process may set or get its own preemption status.

Wait. The invoking process suspends itself for a specified duration or until a specified time occurs. The priority at which the process is to be resumed may also be specified.

Set timeslice. Defines the timeslice quantum (only processes of equal priority are time sliced).

Enable time slicing. Enables the scheduler to perform round-robin, timeslice scheduling.

Disable time slicing. Disable round-robin, timeslice scheduling.

A.5. Communication Management

Send a message. Sends a message from one process to another, without waiting for acknowledgement of message receipt.

Send a message and wait for an acknowledgement. Sends a message from one process to another; the sender blocks while waiting for acknowledgement of message receipt or until an optional timeout expires.

Receive a message. Receives a message from another process, blocking until a message is available or an optional timeout expires.² The Kernel automatically performs any required acknowledgements.

A.6. Interrupt Management

Enable. Allows processing for a specific interrupt to occur.

Disable. Disallows processing for a specific interrupt to occur.

Enabled. Queries whether a specific interrupt is enabled or disabled.

Simulate interrupt. Simulates the occurrence of a specific interrupt in software.

Bind interrupt handler. Asserts that an Ada procedure has been identified as an interrupt handler and is to be executed when the specified interrupt occurs.³

A.7. Time Management

Reset epoch time. Resets the epoch time of the local processor to the specified date/time.

Adjust elapsed time. Increments or decrements the elapsed time of the local processor by the specified amount.

Read clock. Reads the current elapsed time from the local processor clock.

²The sender is not named in the receive primitive; however, the sender can be determined from the message contents.

³A non-Ada code fragment may be used, as long as it follows the semantics of Ada procedures.

A.8. Alarm Management

Set alarm. Sets a timer to expire after the specified duration has elapsed or at the specified time.

Cancel alarm. Cancels an unexpired timer.

A.9. Tool Interface

Begin collection. Begins logging state-change information for the specified process.

End collection. Terminates logging state-change information for the specified process.

Read process table. Copies the Kernel's Process Table into application memory.

Read interrupt table. Copies the Kernel's Interrupt Table into application memory.

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED			1b. RESTRICTIVE MARKINGS NONE		
2a. SECURITY CLASSIFICATION AUTHORITY N/A			3. DISTRIBUTION/AVAILABILITY OF REPORT APPROVED FOR PUBLIC RELEASE DISTRIBUTION UNLIMITED		
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A					
4. PERFORMING ORGANIZATION REPORT NUMBER(S) CMU/SEI-88-TR-17			5. MONITORING ORGANIZATION REPORT NUMBER(S) ESD-TR-88- 018		
6a. NAME OF PERFORMING ORGANIZATION SOFTWARE ENGINEERING INSTITUTE		6b. OFFICE SYMBOL (If applicable) SEI	7a. NAME OF MONITORING ORGANIZATION SEI JOINT PROGRAM OFFICE		
6c. ADDRESS (City, State and ZIP Code) CARNEGIE MELLON UNIVERSITY PITTSBURGH, PA 15213			7b. ADDRESS (City, State and ZIP Code) ESD/XRS1 HANSCOM AIR FORCE BASE, MA 01731		
8a. NAME OF FUNDING/SPONSORING ORGANIZATION SEI JOINT PROGRAM OFFICE		8b. OFFICE SYMBOL (If applicable) SEI JPO	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F1962885C0003		
8c. ADDRESS (City, State and ZIP Code) CARNEGIE MELLON UNIVERSITY SOFTWARE ENGINEERING INSTITUTE JPO PITTSBURGH, PA 15213			10. SOURCE OF FUNDING NOS.		
11. TITLE (Include Security Classification) DISTRIBUTED ADA REAL-TIME KERNEL			PROGRAM ELEMENT NO.		WORK UNIT NO.
			PROJECT NO. N/A		TASK NO. N/A
12. PERSONAL AUTHOR(S) BAMBERGER, COLKET, FIRTH, KLEIN, D., VAN SCOY					
13a. TYPE OF REPORT FINAL		13b. TIME COVERED FROM _____ TO _____		14. DATE OF REPORT (Yr., Mo., Day) AUGUST 88	
15. PAGE COUNT 28					
16. SUPPLEMENTARY NOTATION					
17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)		
FIELD	GROUP	SUB. GR.	ADA, REAL-TIME, KERNEL, DISTRIBUTED SYSTEMS		
19. ABSTRACT (Continue on reverse if necessary and identify by block number) <p>THIS PAPER ADDRESSES TWO DISTINCT NEEDS OF REAL-TIME APPLICATIONS: DISTRIBUTION AND HARD REAL-TIME SCHEDULING MECHANISMS. SPECIFICALLY, THIS PAPER REJECTS BOTH THE NOTION OF MODIFYING THE ADA LANGUAGE TO ACHIEVE NEEDED REAL-TIME SOLUTIONS AND THE CURRENT FAD OF EXTENSIVELY MODIFYING THE ADA COMPILER AND/OR VENDOR-SUPPLIED RUNTIME SYSTEM. INSTEAD, THIS PAPER DEFINES THE FUNCTIONALITY OF A DISTRIBUTED ADA REAL-TIME KERNEL (HEREAFTER CALLED THE KERNEL). THE GOAL OF THE KERNEL IS TO SUPPORT EFFECTIVELY THE EXECUTION OF DISTRIBUTED, REAL-TIME ADA APPLICATIONS IN AN EMBEDDED COMPUTER ENVIRONMENT BY RETURNING CONTROL TO THE USER, WHERE IT BELONGS.</p>					
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED <input checked="" type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS <input checked="" type="checkbox"/>			21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED, UNLIMITED		
22a. NAME OF RESPONSIBLE INDIVIDUAL KARL SHINGLER			22b. TELEPHONE NUMBER (Include Area Code) (412) 268-7630		22c. OFFICE SYMBOL SEI JPO

